PERGAMON Computers and Mathematics with Applications 37 (1999) 37–50

An International Journal
computers & mathematics
with applications

Linear-Time LUP Decomposition of Forest-Like Matrices

D. P. JACOBS
Department of Computer Science
Clemson, SC 29634-1906, U.S.A.
dpj@cs.clemson.edu

V. TREVISAN*
UFRGS-Instituto de Matemática
91509-900 Porto Alegre, RS, Brazil
trevisan@mat.ufrgs.br

(Received May 1998; revised and accepted November 1998)

Abstract—A square matrix M has an LUP decomposition if it can be written as LUP, where L is a unit lower-triangular matrix, U is an upper-triangular matrix, and P is a permutation matrix. We present a linear-time algorithm for finding an LUP decomposition for a nonsingular neighborhood matrix of a tree. We also identify a more general class of matrices we call *forest-like* for which the algorithm can be modified.

Keywords—Graph, Tree, Matrix decomposition, Algorithm.

1. INTRODUCTION

In [1,2], it was shown how to compute the determinant and rank of a tree's adjacency and neighborhood matrices in linear-time, and in [3], a fast method was given for computing the characteristic polynomial of a tree's adjacency matrix. This paper considers LU and LUP decompositions for a class containing these matrices.

Let $G = (V, E)$ be an *undirected graph* with vertices $V = (v_1, \ldots, v_n)$ and edge set E. Edges occur only between pairs of distinct vertices, and between any pair of vertices there is at most one edge. The *adjacency matrix* $A_G = [a_{ij}]$ of G is the $n \times n$ symmetric 0-1 matrix for which $a_{ij} = 1$ if and only if v_i is adjacent to v_j (that is, there is an edge between v_i and v_j). In this paper, graphs are *trees* (i.e., connected, acyclic graphs) or *forests* (i.e., disjoint collections of trees). The *neighborhood matrix* of G, denoted N_G, is obtained by placing 1s along the main diagonal of the adjacency matrix (i.e., $N_G = A_G + I_n$). A square matrix is *lower (upper) triangular* if all entries above (below) the main diagonal are zero. A matrix is *unit* if it has all one's on the main diagonal. A square matrix (over some field) has an *LU decomposition* if there exist a unit lower triangular matrix L and an upper triangular matrix U such that $A = LU$.

A square 0-1 matrix is a *permutation matrix* if every row and every column contain exactly one 1. A square matrix M has an LUP *decomposition* if there exist a unit lower triangular

*Partially supported by CAPES, Brazil, under Grant 0744/96-4.

0898-1221/99/$ - see front matter. Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX
PII: S0898-1221(99)00124-8

matrix L, an upper triangular matrix U, and a permutation matrix P such that $M = LUP$. While there exist nonsingular matrices without LU decompositions, every nonsingular matrix has an LUP decomposition [4].

The significance of these decompositions is that if M can be decomposed to LU then the linear system $Mx = b$ can be solved by first solving $Ly = b$ (say, using forward substitution), and then solving $Ux = y$ (using back substitution). Similar comments apply if M has an LUP decomposition.

In [5], Bevis and Hall gave an algorithm to order the vertices of a tree having a loop at the root, so that its adjacency matrix has an integer LU decomposition. In this paper, we seek LU and LUP decompositions first for neighborhood matrices of trees, and then for a more general class of matrices we call forest-like. Our main result is a linear-time LUP decomposition for M, a nonsingular neighborhood matrix of a tree. We also show how $Mx = b$ can be solved in linear time. This is significant because M^{-1} can be dense, prohibiting its computation in linear-time. Even though we are mainly interested in LUP decompositions, it is convenient to first discuss LU decompositions. In Section 6, we suggest how the LUP decomposition can be extended to a more general class of matrices called forest-like.

2. LU FOR NEIGHBORHOOD MATRICES OF TREES

Given a matrix M, there is a method sometimes used for obtaining an LU decomposition. One starts with M and applies Gaussian eliminations hoping to obtain an upper triangular matrix U. Columns are processed left-to-right, and nonzero entries below the diagonal are eliminated. Each Gaussian transformation is equivalent to multiplying M on the left by a unit lower triangular matrix of the form

$$L_i = \begin{bmatrix} 1 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \vdots & \vdots & \vdots \\ 0 & & & 1 & \cdots & 0 & & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & & & \alpha & \cdots & 1 & & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}. \tag{1}$$

Hopefully, one obtains $L_k \ldots L_1 M = U$. Since

$$L_i^{-1} = \begin{bmatrix} 1 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \vdots & \vdots & \vdots \\ 0 & & & 1 & \cdots & 0 & & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & & & -\alpha & \cdots & 1 & & 0 \\ \vdots & \vdots & \vdots & \vdots & & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}, \tag{2}$$

one can easily compute $M = LU$, where

$$L = L_1^{-1} L_2^{-1} \ldots L_k^{-1}. \tag{3}$$

This method fails when we must process column j, but its diagonal element is 0.

FACT 1. Let $S_1, S_2, \ldots, S_k$ be $n \times n$ matrices of form (1), where S_i contains its subdiagonal nonzero element in column c_i, and $c_1 \leq \cdots \leq c_k$. Then $S_1 S_2 \ldots S_k = (S_1 + \cdots + S_k) - (k-1) I_n$.

PROOF. Straightforward.

We wish to compute (3) easily. Letting $S_i = L_i^{-1}$, suppose S_i has its subdiagonal entry $\beta_i = -\alpha_i$. Assuming the columns are in nondecreasing order, and all β_is occur in different

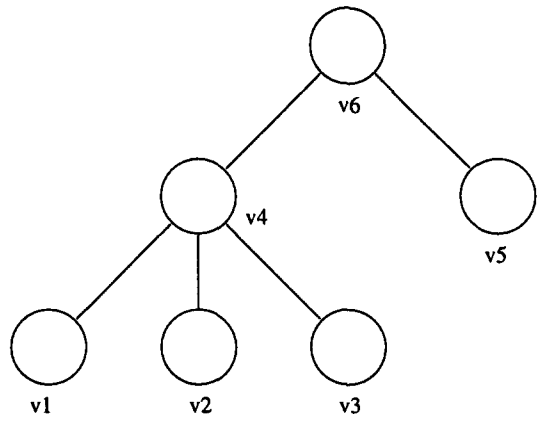

Figure 1.

positions, by Fact 1, the product can be computed by simply storing the β_is into one matrix L. Thus, L is constructed in time proportional to the number of row operations.

Let T be a tree on n vertices. We wish to obtain an LU decomposition for some neighborhood matrix of T. If we use the Gaussian transformation approach described above, the ordering of the vertices will effect the amount of fill-in created by the row operations. We can avoid fill-in by first selecting a root of the tree. We then order the vertices $V = (v_1, \ldots, v_n)$ so that if v_i is a parent of v_j, then $i > j$.[1] For example, consider the tree in Figure 1 whose vertices are ordered as shown. To obtain an LU decomposition for the neighborhood matrix N, we reduce N to upper triangular form, while building the lower triangular product. We maintain two matrices whose product is always N. Initially, we have

$$I_n = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \qquad N = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

After the first elimination, we have

$$L_1{}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}, \tag{4}$$

and after five operations, the first matrix becomes L, and the second becomes U,

$$L_1{}^{-1} \ldots L_5{}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -\frac{1}{2} & 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \end{bmatrix} = U.$$

The vertex order is crucial in preventing fill-in. Because of our vertex order, row operations always represent a child's row acting on a parent's row. Beside the entry being eliminated, the

[1] A postorder satisfies this condition, as do others.

Initialize $L := I_n$
Initialize $d(v_i) := 1$ for all vertices v_i
for $i = 1$ **to** n **do**
  if v_i has a child v_j such that $d(v_j) = 0$ **then**
    return FAIL
  else for each child v_j of v_i
    $d(v_i) := d(v_i) - \frac{1}{d(v_j)}$,
    $L[i,j] := \frac{1}{d(v_j)}$
end loop
return L, $(d(v_1), d(v_2), \ldots, d(v_n))$

Figure 2. LU decomposition of a tree's neighborhood matrix.

only other entry effected is the diagonal entry of the parent. Since our Gaussian transformations do not produce additional fill-in, neither L nor U contain nonzero elements in positions where N is zero. Above the main diagonal, U maintains the original entries of N.

Thus, it is not necessary to store all of U, but only the diagonal. Our algorithm in Figure 2 utilizes this. The algorithm uses two data structures: a matrix L for constructing the lower-triangular factor and the tree T. The vertices v of T are given values $d(v)$ that represent the corresponding diagonal value in the upper triangular factor. The nondiagonal entries in U are implicitly represented by the edges of the tree. As noted, entries in U above the main diagonal do not change. Therefore, U can later be constructed using the diagonal elements $(d_1, d_2, \ldots, d_n)$ returned by the algorithm and the original entries $N_T = [a_{ij}]$ by writing

$$U = \begin{bmatrix} d_1 & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & d_2 & a_{23} & \cdots & a_{2n} \\ 0 & 0 & d_3 & \cdots & a_{3n} \\ \vdots & & & & \vdots \\ 0 & 0 & 0 & \cdots & d_n \end{bmatrix}.$$

Note that vertices are processed bottom-up. Processing a vertex v means that the corresponding diagonal elements are being used to eliminate the 1s representing the edges between v and its children.

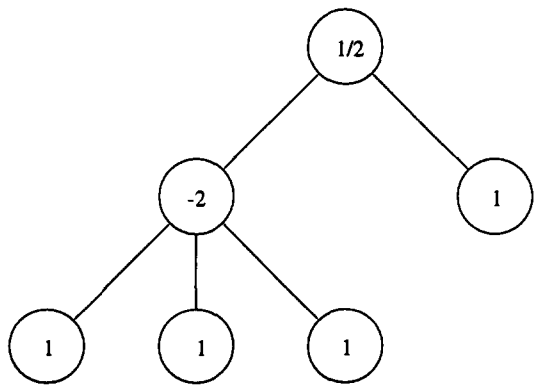

Figure 3. Computing the diagonal of U.

Figure 3 shows the diagonal values on the tree from Figure 1. It may happen that a diagonal element temporarily becomes zero, as in the second matrix of (4). However, our algorithm fails to find an LU decomposition if a vertex, other than the root, is zero *after* being processed.

Figure 4 shows a rooted tree whose neighborhood matrix is singular, and yet our algorithm returns an LU decomposition. This is possible, because the zero diagonal element does not occur

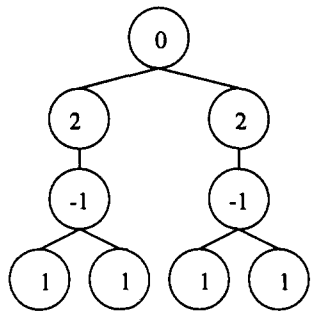

Figure 4. Tree with singular neighborhood matrix having LU decomposition.

until we reach the root. It is interesting to note that if Figure 1's tree is rerooted at vertex v_4, our method fails. Finally, our method fails for the path on four vertices, no matter where it is rooted, yet the neighborhood matrix is nonsingular.

3. LUP FOR NEIGHBORHOOD MATRICES OF TREES

In this section, we describe an algorithm for finding an LUP decomposition of a nonsingular neighborhood matrix of a tree T. Later we will explain how to implement it in linear time. Given T, we choose a root and let $N = N_T$, relative to some ordering satisfying the condition described earlier. Our strategy will be an extension of the method used for the LU decomposition. In particular, we begin with N, and by multiplying on the left by unit lower triangular matrices L_i, and on the right by permutation matrices P_j, we will obtain an upper triangular matrix U. That is,

$$L_k \dots L_1 N P_1 \dots P_l = U.$$

The LUP decomposition is then obtained by

$$N = L_1^{-1} \dots L_k^{-1} U P_l^{-1} \dots P_1^{-1}.$$

As before, the L_is represent Gaussian eliminations. The P_is are permutation matrices that interchange two columns. Thus, $P_i^{-1} = P_i$.

Each L_i^{-1} can be found be replacing α by $-\alpha$ as in (2). To simplify our algorithm, we will insure that any vertex whose diagonal element becomes zero has an index greater than any of its siblings.

Our algorithm, as before, uses the tree T to store the diagonal elements of U and a matrix L which is used to construct (3). We also require a set P of disjoint transpositions. After the diagonal elements are determined and P is known, U can be constructed easily.

Initially, the diagonal values $d(v)$ of all vertices of T are 1, $L = I_n$, and $P = \emptyset$. The algorithm proceeds by processing the vertices of T bottom-up. As in the case of LU decomposition, when a vertex v is to be processed, we look at its children c. If no child of v has a diagonal value of zero, then $d(v)$ is replaced by $d(v) - \frac{1}{d(c)}$ for each child c. That is, $d(v)$ is assigned $1 - \sum \frac{1}{d(c)}$, where the sum ranges over all children. Moreover, for each child v_j of v_i, we let $L[i,j] = \frac{1}{d(v_j)}$. The set P is not modified.

On the other hand, what happens if $d(c) = 0$ for some child c of v? (When this happens in our LU decomposition method, the algorithm fails!) It is important to realize that since N is nonsingular, there can be at most *one* child of v whose value is 0. It is also important to realize that, by our assumption, the zero diagonal element occurs after all diagonal elements of its siblings, as in Figure 5.

We *first* process the nonzero children w of v in the normal way. That is, $d(v)$ is replaced by $d(v) - \frac{1}{d(w)}$ for every child $w, d(w) \neq 0$. Our strategy is to *next* interchange the columns of c and v, obtaining the matrix in Figure 6. *Next*, if necessary, we use the 1 in the row of c to

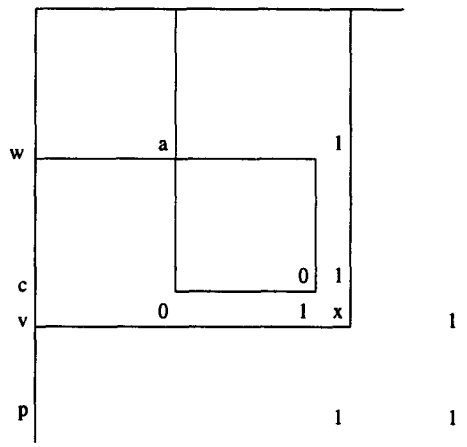

Figure 5. Vertex v and child c with $d(c) = 0$.

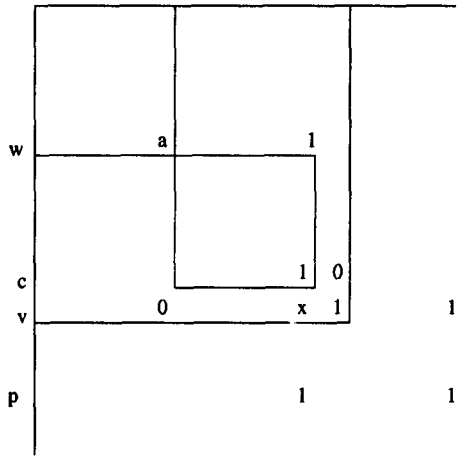

Figure 6. Matrix with columns interchanged.

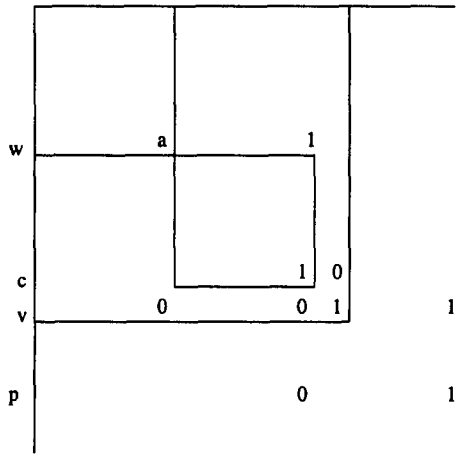

Figure 7. After two row operations.

annihilate the entry below it in the row of v. *Finally*, if v is not the root, we use the 1 in the row of c to annihilate the entry in the row of p (shown in Figure 6), where p is the parent of v. By our assumption on the vertex ordering there are no other nonzero entries below the 1 in row c. The operations produce no fill-in and are recorded in L. The result is depicted in Figure 7.

The reason that the row of c and not the row of v is used to annihilate the 1 in row p is that v's entry x could be zero. Moreover, v's row would produce undesirable fill-in. Since this 1

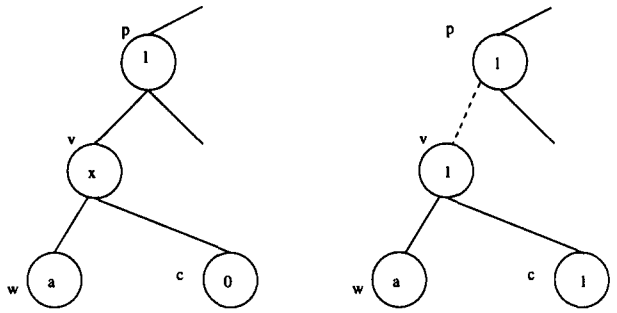

Figure 8. Tree representation before and after the operation.

is now eliminated early, we will not need vertex v when we later process vertex p. Therefore, we no longer consider v a child of p. We represent this by removing the edge between v and p. In summary, the left picture in Figure 8 represents Figure 5, and the right picture in Figure 8 represents Figure 7.

Assuming $c = v_j$, $v = v_i$, and $p = v_k$, the set P is given the transposition (i, j), and the matrix L is modified by

$$\begin{aligned} L[i,j] &= d(v), && \text{if } d(v) \neq 0, \\ L[k,j] &= 1, && \text{if } v \text{ has a parent } p. \end{aligned}$$

Figure 9 depicts the formal description of the algorithm in which L, P, and the diagonal elements of U are constructed. The remainder of U can be constructed with the algorithm shown in Figure 10.

INPUT. A rooted tree T having a nonsingular neighborhood matrix. We assume vertices have been post-ordered, and "zero" vertices have greater index among siblings.

for $i = 1$ **to** n **do**
  for each child v_j of v_i where $d(v_j) \neq 0$ **do**
    $d(v_i) := d(v_i) - \frac{1}{d(v_j)}$,
    $L[i,j] := \frac{1}{d(v_j)}$
  end loop
  if v_i has a child v_j with $d(v_j) = 0$ **then**
    $P := P \cup \{(i,j)\}$
    if $d(v_i) \neq 0$ **then**
      $L[i,j] := d(v_i)$
    end if
    if v_i has a parent v_k **then**
      remove the edge between v_i and v_k
      $L[k,j] := 1$
    end if
    $d(v_i) := 1$
    $d(v_j) := 1$
  end if
end loop

Figure 9. LUP decomposition: construction of L and P.

We summarize the LUP method. Given a tree T, we first choose an arbitrary root. The vertices are then postordered. If necessary, the order is adjusted so that any zero vertex is ordered after

its other siblings. Next, we construct L, P, and the diagonal values of U using the method of Figure 9. Finally, we construct the remaining entries of U using Figure 10's method.

The following example illustrates our LUP decomposition algorithm. Consider the tree in Figure 11. The numbers beside each vertex show the vertex ordering. The corresponding neighborhood matrix is

$$N = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}. \tag{5}$$

The values in the vertices are U's diagonal values obtained by our algorithm. Note that when processing vertex 9, we must apply the operation of Figure 8 in order to deal with the child (vertex 8) whose value is zero. Thus, according to the operation, both diagonal values become 1 and the edge is deleted. The transposition is $(8, 9)$ is recorded. We obtain the matrices shown in Figure 12.

$U := N_T$
for each $(i, j) \in P$, $i > j$
    exchange columns i and j in U
    $U_{ji} := 0$
end loop
for $i = 1, \ldots, n$
    $U_{ii} := d(v_i)$
end loop
$U_{ij} := 0$ for $i > j$

Figure 10. LUP decomposition: construction of U.

4. LINEAR-TIME IMPLEMENTATION

Traditional methods for obtaining an LUP decomposition require $O(n^3)$ time, while more efficient methods exist whose running time is comparable to $O(n^{2+\epsilon})$ fast matrix multiplication algorithms (see [4, Theorem 6.4]). In describing our LU method, we have used an $n \times n$ matrix L in which we construct the unit lower triangular factor. Each column of this matrix has at most one nonzero entry α below the main diagonal. Therefore, L can be implemented in $O(n)$ space as a $2 \times n$ array A in which $A[1, j] = i$ and $A[2, j] := \alpha$ mean that $L[i, j] = \alpha$. In the case of our LUP decomposition, the unit lower triangular matrix L has at most two nonzero entries below

the main diagonal, and can be represented in a similar way. The tree T can be implemented using conventional node and pointers methods. Each vertex should have a pointer to its parent. In order to represent a "broken" edge, an additional field in the record can be used. After Figure 9's algorithm has finished, the factor U may be constructed in linear-time as in Figure 10 using standard sparse matrix techniques.

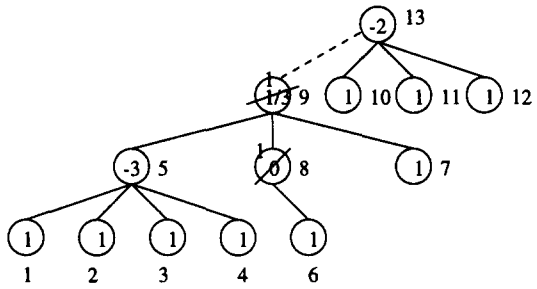

Figure 11. Example.

5. SOLVING $Nx = b$ IN LINEAR-TIME

Suppose we are given a nonsingular neighborhood matrix N for a tree, a column vector b, and we wish to solve $Nx = b$. If N is represented using a sparse data structure, then we can solve the equation in linear time. To do this, we first construct the tree T. Next, we choose an arbitrary root, and postorder the vertices using the constraints described earlier. Letting N' be the neighborhood matrix under this ordering, we may construct a permutation matrix Q such that $N' = QNQ$. Next, we solve $N' = LUP$. Then the solution to our original equation is given by $LUPQ^{-1}x = Qb$. Note Q^{-1} can be computed in linear time.

6. ALGORITHMS FOR FOREST-LIKE MATRICES

In this section, we consider a class of sparse matrices that contain the neighborhood matrices of forests. An $n \times n$ matrix $B = [b_{i,j}]$ is *forest-like* (*tree-like*) if there exists a forest (tree) with an $n \times n$ neighborhood matrix $A = [a_{i,j}]$ in which $b_{ij} \neq 0$ only if $a_{ij} \neq 0$. Consider the matrices

$$B = \begin{bmatrix} 4 & 0 & -1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ -2 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix}, \qquad A = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

A is the neighborhood matrix of the forest of Figure 13, and so B is forest-like. Given a sparse representation for an $n \times n$ matrix B, we can decide in linear-time if B is forest-like: if B has more than $3n - 2$ nonzero entries, it cannot be forest-like. Otherwise, construct a matrix A in the following way: first replace all nonzero entries by 1. Next, if $a_{i,j} = 1$ and $a_{j,i} = 0$, replace $a_{j,i}$ by 1. Let $a_{i,i} = 1$, for all i. Now check that this is the neighborhood matrix for an acyclic graph.

A forest-like matrix B is represented by first constructing the associated forest, and then rooting the trees. Each diagonal element $b_{i,i}$ of B is the label of the corresponding vertex v_i. If v_i is a child of v_j, the edge between v_i and v_j is labeled by $(b_{j,i}, b_{i,j})$. The matrix B above is represented by the forest of Figure 14.

We now explain how $\det(B)$ can be computed. Given a forest-like matrix B, it is easy to see that for some permutation matrix P, we have

$$PBP = \begin{bmatrix} B_1 & 0 & 0 & 0 \\ 0 & B_2 & 0 & 0 \\ \vdots & & & \vdots \\ 0 & 0 & 0 & B_k \end{bmatrix},$$

where each $B_i, i = 1, \ldots, k$ is a *tree-like* matrix. As it is well known from the theory of block matrices,

$$\det B = \det B_1 \ldots \det B_k.$$

So for the computation of the determinant, we consider only tree-like matrices. In [1], an algorithm was given to compute the determinant of matrices $B + \lambda I$, where B is the adjacency matrix of a tree. We now modify the method for tree-like matrices. The algorithm works directly on the tree representing the matrix. It may be seen as a diagonalization process of the matrix.

$$L = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -\frac{1}{3} & 0 & 1 & \frac{1}{3} & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1
\end{bmatrix}$$

$$U = \begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -3 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2
\end{bmatrix}$$

Figure 12. LUP decomposition of (5).

$$P = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

Figure 12. (cont.)

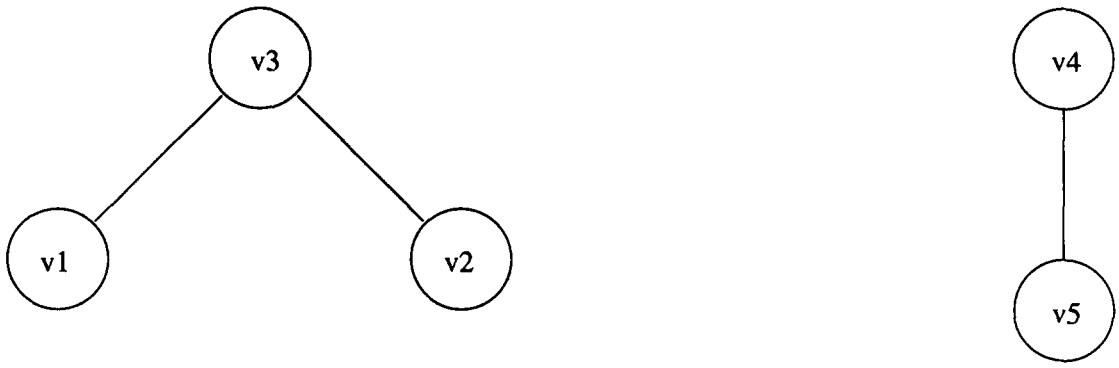

Figure 13. Forest represented by matrix A.

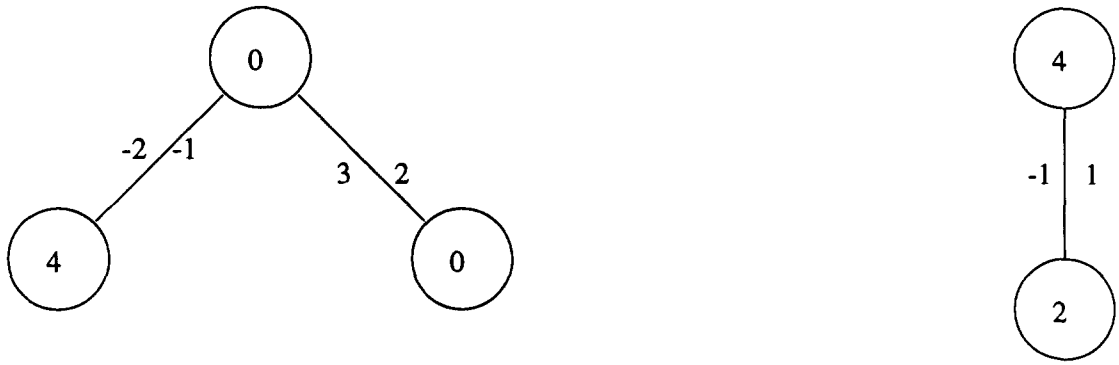

Figure 14. Representation of matrix B.

The algorithm proceeds bottom-up. The diagonalization is obtained by two kinds of operations. When all the diagonal elements of the children of a vertex are nonzero, then those elements are used to eliminate the off-diagonal nonzero entries using ordinary row and column operations.

Initialize $d(v_i) := b_{i,i}$ for all vertices v_i

Initialize $U := V$ (all vertices undeleted)

Initialize all leaves "processed" and all non-leaves "unprocessed"

Initialize $\delta = 0$

Initialize $e = 1$

while unprocessed $\neq \emptyset$ **do**

choose any unprocessed v_i of greatest depth

unprocessed = unprocessed $-\{v_i\}$

if v_i has undeleted child with value 0 **then**

select such a child v_j

$U := U - \{v_i, v_j\}$

$\delta := \delta + 1$

$e := e \cdot b_{i,j} \cdot b_{j,i}$

else $d(v_i) := d(v_i) - \sum_{v_j \in C} \frac{b_{i,j} b_{j,i}}{d(v_j)}$,

where the sum ranges over undeleted children c of v.

end loop

$\det = (-1)^{\delta} \cdot e \cdot \prod_{u \in U} d(u)$

return

Figure 15. Computing $\det(B)$ for tree-like matrix.

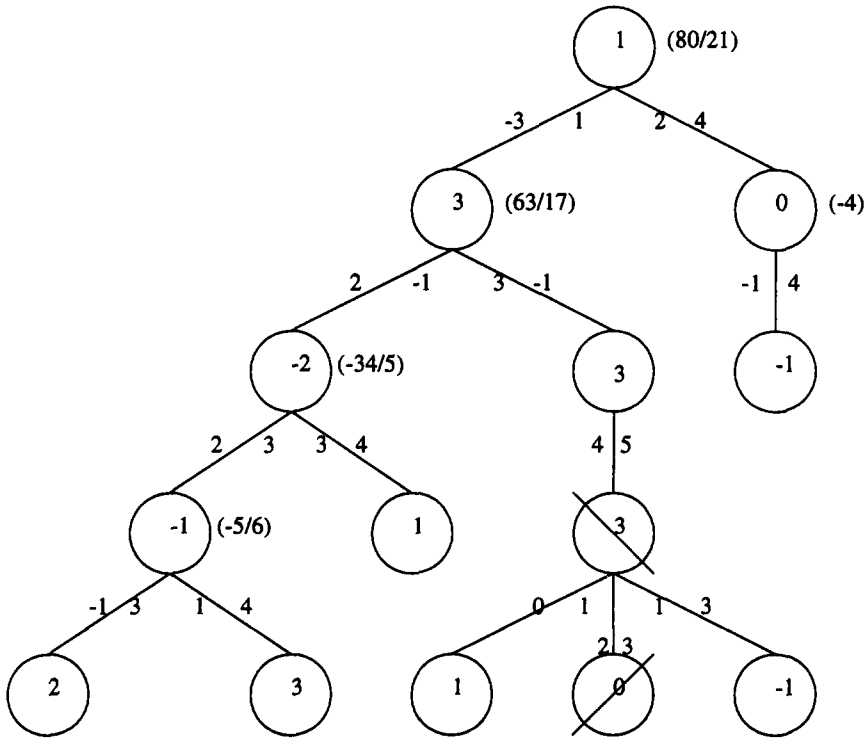

Figure 16. Computing the determinant of matrix A in (7).

But when one child has a zero diagonal element, we use Laplace expansion for the determinant and delete the row and column of one of the off-diagonal nonzero entries. We then use Laplace expansion again to delete the row and column of the other off-diagonal nonzero entry. This pair of deletions introduces a sign change.

The algorithm associates each vertex v_i with a value $d(v_i)$, that is initialized as $d(v_i) = b_{i,i}$. A variable δ (for deletions) is initialized to zero and a variable e (for expansions) is initialized to 1. The algorithm processes the vertices bottom-up, beginning with the leaves, which are initially declared processed. In general, it chooses any unprocessed vertex v_i of maximum depth, marks it processed and the following is executed. If there exist undeleted children v_j of v_i for which $d(v_j) = 0$, then one such child is selected. Both v_i and v_j are deleted, δ is incremented by one, and e is multiplied by the weight values $b_{i,j}$ and $b_{j,i}$ of the edge between v_i and v_j. All other children are uneffected, including those that might have zero values. If, on the other hand, all undeleted children of v_i have nonzero values, and C is the set of (undeleted) children of v_i, then $d(v_i)$ is replaced by

$$d(v_i) - \sum_{v_j \in C} \frac{b_{i,j} b_{j,i}}{d(v_j)}.$$

After all the vertices are processed, we compute

$$\det = (-1)^{\delta} \cdot e \cdot \prod_{u \in U} d(u), \tag{6}$$

where U is the set of undeleted vertices. The algorithm is summarized in Figure 15.

We now compute the determinant of the tree-like matrix

$$A = \begin{bmatrix}
2 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 3 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 2 & 1 & 0 & 0 & 3 & 0 & 5 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 2 & 3 & 0 & -2 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 3 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 4 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 3 & 0 & 3 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 2 & 1
\end{bmatrix} \tag{7}$$

represented in Figure 16. The numbers shown inside each vertex are A's diagonal elements. The pairs of numbers on the edges are the off-diagonal elements of A. The values in parenthesis outside each vertex are new diagonal values computed as $d(v) - \sum_{c \in C} \frac{v_r \cdot v_c}{d(c)}$. These replace the old diagonal values. Notice that there is one deletion, so the value of δ is 1, meaning that the sign changes once. The expansion variable e gets the value of the row weight times column weight of the edge between the two vertices deleted ($2 \cdot 3 = 6$). To compute the determinant, we multiply

$(-1)^6 \cdot e = -6$ by the product of all final diagonal values of undeleted vertices in the tree, as in equation (6). The matrix has determinant 34560.

Clearly, the number of operations performed by the algorithm of Figure 15 is linear in n. So for forest-like matrices, the determinant can be computed in linear time.

The ideas presented in this section can be modified to provide matrix factorization. In particular, the LUP factorization of a nonsingular forest-like matrix can be found in $O(n)$ time. Given a nonsingular forest-like matrix M, the solution to $Mx = b$ can be computed in linear time by first decomposing M into LUP. This is faster than computing M^{-1} because this matrix can be dense. Indeed, let T be the tree in which the root is adjacent to every other vertex. Experiments show that the inverse of the neighborhood matrix contains no zeros.

An interesting research topic is to study the class of matrices that can be decomposed into a small product of forest-like matrices. Given the decomposition for such a matrix M, $Mx = b$ could be solved in linear time.

REFERENCES

1. G.H. Fricke, S.T. Hedetniemi, D.P. Jacobs and V. Trevisan, Reducing the adjacency matrix of a tree, *The Electronic Journal of Linear Algebra* **1**, 34–43, (1996).
2. D.P. Jacobs and V. Trevisan, The determinant of a tree's neighborhood matrix, *Linear Algebra and Its Applications* **256**, 235–249, (1997).
3. D.P. Jacobs and V. Trevisan, Constructing the characteristic polynomial of a tree's adjacency matrix, *Congressus Numerantium* (to appear).
4. A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, (1974).
5. J.H. Bevis and F.J. Hall, Integer LU factorizations, *Linear Algebra and Its Applications* **150**, 267–285, (1991).